Coalescing and Deduplicating Incremental Checkpoint Files for Restore-Express Multi-Level Checkpointing

Purushottam Sigdel and Nian-Feng Tzeng, *Fellow, IEEE*

Abstract—In multicore systems, a large portion of checkpoint time overhead can be hidden from the execution critical path by resorting to a dedicated checkpointing thread run concurrently with regular execution threads for compressing checkpoint files to lower checkpointing overhead. On the other hand, the restore time is on the critical path that cannot be hidden, making it most important to accelerate execution restore upon failures. This work pursues a restore-express (REX) strategy for multi-level checkpointing (MLC), applicable to any incremental checkpointing (IC). Oblivious to application codes, REX employs adaptive IC (AIC) for local (L1) checkpointing and follows our runtime control for second-level (L2) checkpointing, with its aim at express restore from failures while holding down the overall execution time. It takes advantage of two unique insights for overhead reduction: (1) the modified pages of an incremental checkpoint file are likely to exist in a subsequent checkpoint file, and (2) many data patterns (on an average, some 40 percent of them) stay unchanged from one L2 checkpoint file to the next. These insights enable REX to (1) coalesce IC files (by involving only the last copy of every dirty page among files) and (2) boost file compression across multiple L2 checkpoints. Time and storage overhead results of REX during normal job execution are gathered for 16 benchmarks from SPEC, PARSEC, and NPB suites. The evaluation outcomes of the execution restore time confirm that REX is fast and able to quicken restore by a factor of 4.5× when compared with its IC counterpart (without utilizing the unique insights), while incurring same execution time overhead.

Index Terms—Compression and deduplication, execution restore, hash functions, incremental checkpointing, multi-level checkpointing

1 Introduction

Checkpointing permits job execution recovery from failures, by recording the execution state of a running job. It typically requires suspending job execution in order to take the execution state, involving time overhead. Checkpoint files are kept in storage for later recovery use when needed, and they involve storage overhead. Overhead in time and storage due to checkpointing depends largely on the checkpoint file sizes and the checkpoint frequency, which should be kept as low as possible.

Full checkpointing (FC) takes the complete memory footprint, typically involving high overhead in time and storage but permitting simple job execution restore via the latest FC file. Incremental checkpointing (IC) [7], [8], on the other hand, is commonly adopted to reduce the checkpoint data volume, by saving only modified and new (due to dynamic allocation) memory pages into the checkpoint. Furthermore, IC with an adaptive checkpoint frequency has been pursued for data volume reduction. It can be achieved by determining appropriate points of checkpointing time through cost prediction on-the-fly [8], [9], omitting certain durations without checkpointing (based on the expected recovery time) [10], or embracing varied block boundaries optimally (instead of fixed page boundaries) [11]. Recent adaptive IC (AIC) relies on effectively predicting the execution state similarity degree to determine the enticing points of time to take checkpoints, arriving at smallest possible checkpoint files [12].

A networked system may keep checkpoint files not only in local storage but also in remote shared storage to ensure acceptable reliability for long-running jobs, called multi-level checkpointing (MLC) [3], [4]. With local (L1) checkpointing and remote (say, L2) checkpointing, MLC is necessary in order to tolerate various failure/unavailability types, including ones that render local storage unavailable. MLC is relatively expensive due to involving data transfer (over the network) and data writes (to remote storage) for L2 checkpointing, desiring checkpoint data volume reduction. To contain overhead and avoid bottlenecks, MLC usually keeps all checkpoint files in (less expensive) local storage while sending only a few checkpoints to (shared and resilient) remote storage.

Execution state restore from failures is on the critical path, making it utterly important to shorten the restore time. To this end, we address restore-express (REX) checkpointing for MLC, with its L1 checkpointing following AIC [12] and its L2 checkpointing governed by our runtime control which aims at express restore from failures while holding down the overall execution time. Typically, a few L1

• *The authors are with the School of Computing and Informatics, the University of Louisiana at Lafayette, 301 E. Lewis Street, Lafayette, LA 70504. E-mail: {pxs9444, tzeng}@louisiana.edu.*

Manuscript received 15 Sept. 2017; revised 12 May 2018; accepted 31 May 2018. Date of publication 5 June 2018; date of current version 9 Nov. 2018. (Coressponding author: Purushottam Sigdel).
Recommended for acceptance by T. Kosar.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2844210

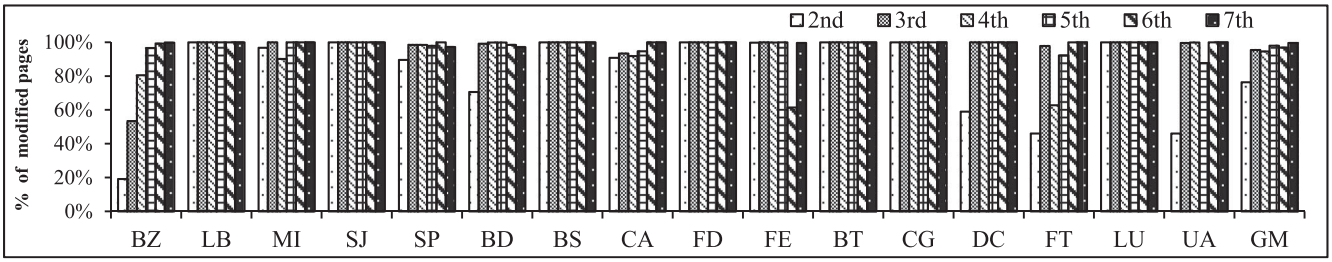

Fig. 1. Percentage of pages in j^{th} incremental checkpoint file (obtained by AIC [12]) that had been modified in *any* prior (i^{th}, with $i < j$) checkpoint under various benchmarks. (For example, the 4^{th} IC of BZ has $>$ 80 percent of its pages existing in prior IC files).

checkpoints are taken before conducting one L2 checkpointing at the desirable time point. Hence, L2 inter-checkpointing intervals vary. REX achieves exceeding overhead reduction by (1) coalescing batches of IC files before checkpointing them in L2 storage and (2) deduplicating data patterns both within each L2 checkpoint and across multiple L2 checkpoints, accelerating restore from failures. File coalescence combines a sequence of IC files in each batch by skipping repetitive dirty pages present in previous files, retaining only the last copies. Coalescence effectiveness stems from our first insight, demonstrated in Fig. 1. From the figure, the majority of pages in the k^{th} IC file, for $k \geq 3$, are seen to exist also in prior IC files for various benchmarks from PARSEC [20], SPEC CPU2006 [19], and NAS Parallel Benchmarking [18] suites (which are detailed in Table 2 of Section 5.2). As a result, REX lowers L2 checkpointing overhead effectively via coalescing IC files in every batch of L2 checkpoints, to arrive at one coalesced L2 checkpoint file (called an L2 checkpoint file for short). This insight also makes it possible for REX to quicken execution restore from transient failures using checkpoint files kept in L1 storage.

In addition, REX employs page-aware deduplication, which compresses pages independently using common data patterns recorded not only within a given coalesced L2 checkpoint file but also across multiple L2 checkpoint files to significantly reduce the checkpoint data volume, with a fixed pattern length (say, 32 bytes). This is made possible by leveraging the second unique insight of L2 checkpoint files, as shown in Fig. 2. According to the figure, modified pages, on an average, have some 40 percent of their data patterns (sized with 32 bytes) identical to those of corresponding pages in the prior L2 checkpoint file, under various benchmarks. Reducing checkpoint data volume shortens file read and transfer times from L2 (or L1, if available) storage for faster restore, besides lowering file transfer times to L2 storage when taking checkpoints.

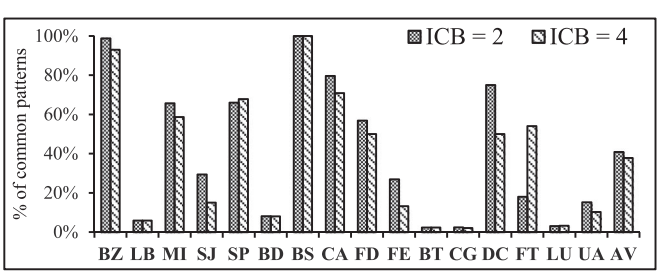

Fig. 2. Percentage of data patterns in a modified page at the j^{th} L2 checkpoint file repeated in the corresponding page at the $(j+1)^{\text{th}}$ L2 checkpoint files under various benchmarks, for L2 inter-checkpointing batch with 2 ($\text{ICB} = 2$) and 4 ($\text{ICB} = 4$) IC files.

REX is the very first MLC that checkpoints IC files to L2 storage, instead of FC (full checkpoint) files like the only earlier known MLC with IC [38]. Under the multicore system, REX incurs negligible execution time overhead by utilizing a dedicated thread to handle L2 checkpointing activities. In fact, for our testbed (detailed in Section 6) with mean time between failure (MTBF) of 10,000 seconds, REX exhibits the average execution slowdown, according to Fig. 11, by less than 3.0 percent (or by $<$ 2.5 percent) under four cores (or eight cores) per node when compared with execution without checkpointing and without failures at all. Note that REX is to have a shorter overall exection time than its counterpart without any checkpointing, if failures exist during execution.

The contribution of this paper is four-fold: (1) it is the very first MLC that checkpoints IC files to L2 storage; (2) it proposes a strategy of coalescing incremental checkpoint files to minimize the overall execution time; (3) it introduces a page-aware compression technique that permits decompressing pages independently; (4) it implements and evaluates REX extensively using three benchmark suites. Evaluation results demonstrate that REX accelerates the mean restore time by a factor of 4.5×, when compared with its earlier counterpart (i.e., a typical IC). In addition, REX enjoys 4.1× faster in restore than the recently pursued mechanism of Libhash [33]. Aggregative coalescence and effective deduplication make REX especially attractive for jobs with large memory footprints.

The rest of the paper is organized as follows. Section 2 provides background, and the proposed REX design is outlined in Section 3. Section 4 extracts the L2 checkpointing characteristics, while Section 5 unveils our evaluation methodology, and Section 6 presents simulation results and discussion. Related work is presented in Section 7, with conclusion given in Section 8.

2 Background and Motivation

2.1 Checkpointing

Various checkpointing mechanisms have been proposed to reduce storage and time overheads associated with checkpointing [6], [7], [8], [10], [28], [39]. To this end, incremental checkpointing (IC) has been commonly adopted [7], [8], [11], [12], [30], [33] for overhead reduction, by saving only modified and new memory pages into the checkpoint files. As a quantitative evidence, we measured the normalized IC size ratio (i.e., the IC file size divided by the full checkpoint (FC) file size) of various benchmarks under AIC [12] for single level (L1) storage, depicted in Fig. 3. The IC size ratio

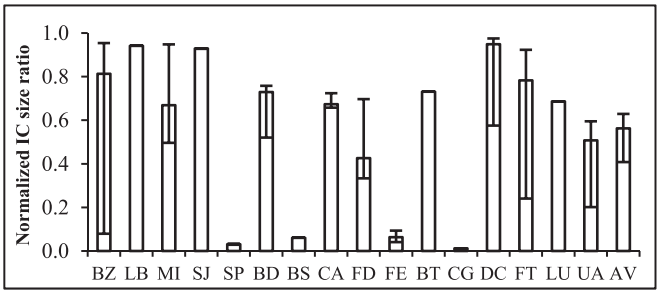

Fig. 3. IC file size ratio, normalized with respective to the corresponding FC size, for various benchmarks.

distribution for each benchmark is also shown inside its corresponding bar. Four (SP, BS, FE, and CG) out of sixteen benchmarks demonstrate their mean IC sizes to stay below 7 percent (of their FC counterpart sizes). Five benchmarks (BZ, MI, FD, FT, and UA) have high (>25 percent) variances (defined as the standard deviation divided by the mean), as can be seen in Fig. 3. On an average, incremental checkpointing cuts the checkpoint data volume by some 45 percent, as shown by the rightmost bar in the figure, with its IC size ratio varying by about 12 percent. This work further explores incremental checkpointing under MLC, besides its shown checkpoint size reduction under single level storage.

2.2 Compression and Deduplication

Compression typically refers to data size reduction within one file (e.g., the compression library of bzip2 [21]), whereas deduplication refers to size reduction over multiple files (e.g., Xdelta3 [14]). A freely available and high-quality data compressor based on a block-sorting lossless compression algorithm developed earlier, bzip2 is popularly applied to compress files [21]. On the other hand, Xdelta3 [14] is a delta compressor that hashes a block of source data and uses the hash table to identify its longest match to the set of target data, originally based on Rsync algorithm [17]. This article uses the terms of compression and deduplication interchangeably.

2.3 Execution State Restore

In general, the latency of execution state restore under MLC is contributed by (1) checkpoint file reads from L1/L2 storage, (2) file transfer over the network to a substitute node (where execution resumes therein), and (3) decompressing the checkpoint file(s), if they are stored in a compressed form. With FC, job execution can be restored from failures simply by the latest checkpoint in either the first level (L1) storage, if accessible, or high level (say, L2) storage. Under IC (incremental checkpointing), however, execution restore typically involves all incremental checkpoint files plus the first full checkpoint. Hence, the one (and only one) MLC with IC [38] pursued so far kept its incremental checkpoints only in L1 storage while holding full checkpoints (FCs) in L2 storage. In contrast, this work pursues IC under MLC effectively by keeping coalesced and compressed IC files in L2 storage.

3 REX Checkpointing Design

REX employs AIC [12] for L1 checkpointing which supports incremental checkpointing via the Unix mprotect() system call for collecting the list of dirty pages during each

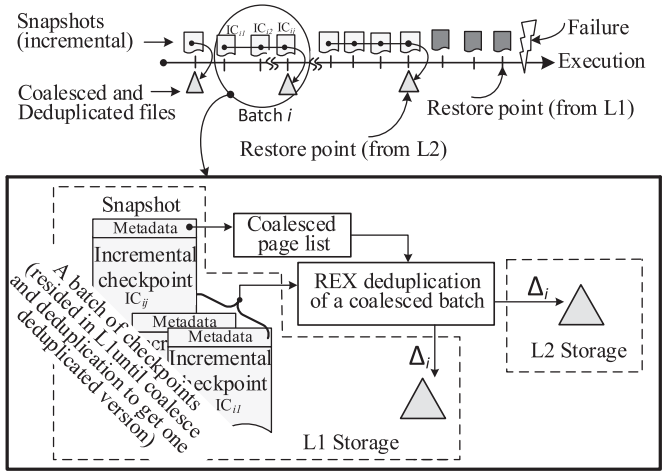

Fig. 4. REX coalescence and page-aware deduplication, with each Δ_i denoting a coalesced and deduplicated version obtained from the ith batch of incremental checkpointing (IC) files, termed $\mathrm{IC}_{i1}, \mathrm{IC}_{i2}, \ldots, \mathrm{IC}_{ij}$. (For clarity, only the details of Batch *i* are shown inside the bold box).

checkpoint interval. The mprotect() system call lets a program set its memory page protection from writing. If the program attempts to modify such a page, the page-fault signal is raised and caught by the signal handler. At the beginning of each checkpoint interval, AIC write-protects target pages in entire process address space. Each first writing attempt to a protected page (1) triggers the signal handler to add the page to the dirty page list and (2) unprotects the page. AIC kernel module then uses this dirty page list to write the modified pages into the L1 checkpoint. Following a set of L1 checkpoints, REX takes an L2 checkpointing, with its aim at express restore from failures and the overall execution time near the lowest possible. Oblivious to application codes, REX takes advantage of two unique insights we have observed for reducing time and storage overhead: (1) the modified pages of an incremental checkpoint file are likely to exist in a subsequent checkpoint file, and (2) many data patterns (on an average, some 40 percent of them) stay unchanged from one L2 checkpoint file to the next. These insights permit REX to (1) coalesce IC files (by involving only *the last copy of every dirty page* among files) and (2) achieve superior deduplication for data patterns not only within the current coalesced L2 checkpoint file but also across earlier L2 checkpoint files. Employing fixed data pattern length (say, 32 bytes) to reduce checkpoint data volume, REX shortens the times of data reads (from storage) and data transfer (to the target node where execution resumes therein) after failures occur to achieve express execution state recovery.

3.1 REX Operational Overview

Implemented on top of AIC, REX performs a full checkpoint (FC) in its very first checkpointing. Such an FC file, after deduplication, is saved into both (L1 and L2) storages. REX then takes incremental checkpoints (ICs) several times locally to produce a batch of IC files before taking one L2 checkpointing, governed by our runtime control. To lower the L2 checkpoint data volume, the batch of IC files (say, $\mathrm{IC}_{i1}, \mathrm{IC}_{i2}, \ldots, \mathrm{IC}_{ij}$ in Fig. 4) is coalesced and then deduplicated before transferred to L2 storage.

As shown in Fig. 4, REX stores the IC files of every batch, say Batch i, in local (L1) storage until they are replaced by one

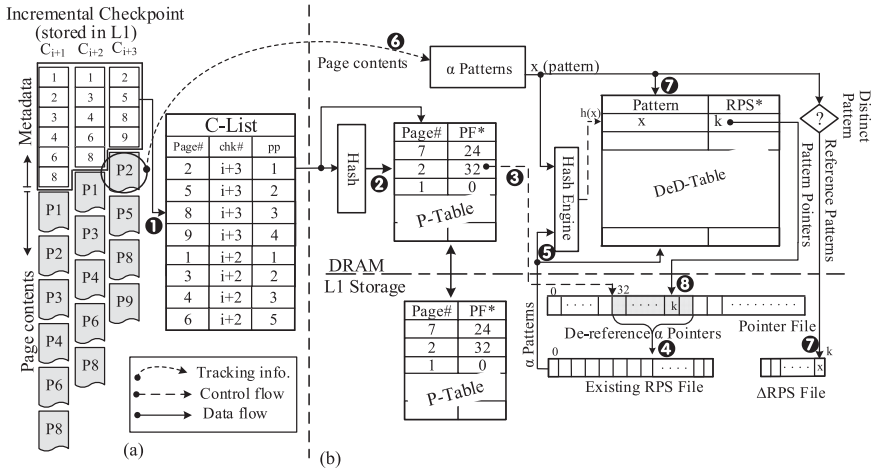

Fig. 5. Coalescence and dedpulicaton of a batch of incremental checkpoint files under REX. ❶ Page Coalescer (PC) processes the header of a batch of checkpoint files and generates the coalesced list (known as C-List) of the distinct pages in (a). Maintaining a list of involved pages and their boundaries through a page table (P-Table), Deduplication Unit (DU) becomes page-aware (b). ❷ Being page-aware, DU takes a page number from C-list and searches a matching page in P-table using hash. ❸ For a matched page, DU reads out α pointers from Pointer File (PF), starting at index PF* and ❹ dereferences them from existing Reference Pattern Sequence (RPS) File. ❺ With the help of hash engine, the dereferenced α patterns (i.e., prior version of page contents) are inserted into the De-duplication table (DeD-Table), and ❻ the new page contents is loaded for deduplication. ❼ Unique pattern, x, enters the DeD-Table and Δ RPS File at location k, ❽ while the corresponding pointer in PF is updated with k. At the end of each batch, REX produces PF, Δ RPS File, and P-Table, which then replaces original IC files in L1 storage and are also transferred to L2 storage for multi-level checkpointing (MLC).

coalesced page-aware deduplicated version (denoted by Δ_i). The coalesced and deduplicated version of Batch i is then kept in L2 storage as an L2 checkpoint. If a failure occurs, REX restores job execution from its checkpoints either from multiple Δ's kept in L2 storage (for a permanent failure) or from multiple Δ's and IC files kept in L1 storage (for a transient failure). To achieve page-aware compression, REX deduplication module employs the fixed sized block (say, 32 bytes) mapped into a 4-byte pointer (instead of 16-to-20 bytes fingerprints [33]). By exploiting the second insight (see Fig. 2), REX thus reduces the checkpoint data volume significantly. After a failure happens, REX, being page-aware, utilizes mapping pointers associated with each involved page to restore the job expressly.

Here, L2 storage is assumed to be failure-free. A failure is either transient (recoverable locally from the latest L1 checkpoint(s)) or permanent (to result in total node failures). After a permanent failure, the application running on a failed node needs to restart on a substitute node, making use of the latest L2 checkpoint(s).

3.2 REX Coalescence and Deduplication

L2 checkpointing involves two major components: Page Coalescer (PC) and Deduplication Unit (DU), as illustrated in Figs. 5a and 5b, respectively. For each batch of incremental checkpoints (C_i's), DU produces *PF* (Pointer File), Δ*RPS* (Reference Pattern Sequence) File, and *P-Table* for replacing original C_i's in L1 storage and checkpointing in L2 storage as well, to realize MLC. It should be noted that the REX deduplication process during a job execution involves local storage only, and that it updates the remote storage using related files only at the end of deduplication.

3.2.1 Page Coalescer (PC)

PC gathers the distinct pages from a batch of n IC files, by skipping repeated pages in earlier checkpoints via examining the header of every IC file, as depicted in Fig. 5a. It drops repeated pages in earlier IC files without processing them and collects all distinct pages to constitute the coalesced list, called C-List, as shown in Fig. 5a. There are only 8 distinct pages in C-List, out of 15 involved in the batch of three IC files. Each list entry registers three attributes: (1) the page number, i.e., page#, (2) the IC file ID, i.e., chk#, and (3) offset position where that page is resided in the IC file, i.e., page position (or pp for short).

3.2.2 Deduplication Unit (DU)

Once PC creates a complete *C-List*, DU starts effective deduplication by breaking the page contents into fixed-size data blocks (also called patterns in this paper), and all distinct blocks are kept in RPS File, to which the data blocks of any page (say, 4K bytes) are pointed, as shown in Fig. 5b. DU deduplicates every page by examining its contents one block after another, with each block replaced by the pointer which denotes the position where the examined block is located in RPS File. The page, after its deduplication, is denoted by a collection of α ($= 4\mathrm{K\,bytes}$ / data block size, in bytes) pointers. Accordingly, a checkpoint file with P pages is represented by $(P \times \alpha)$ pointers, which constitute PF (i.e., Pointer File) for the checkpoint.

DU employs P-Table (i.e., Page-Table) to record all involved pages. Each page recorded in P-Table consists of two fields, as shown in Fig. 5b: (1) page number, i.e., Page#, and (2) PF pointer to denote the start position of α consecutive reference pointers resident in Pointer File, i.e., PF*. The need of PF pointers is to accommodate new pages that can be created dynamically during job execution. The page number and PF pointer are distinct and fixed throughout the page-aware deduplication process.

3.2.3 Deduplication Implementation

DU hashes every page number existing in C-List via a hash function h to produce an index for one access to P-Table. A

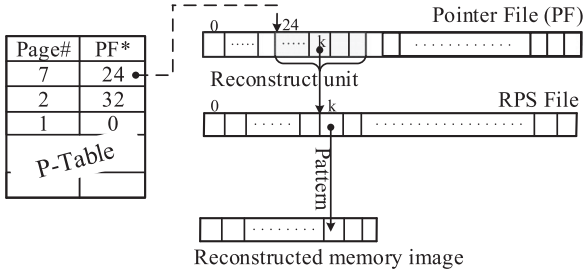

Fig. 6. REX restoration using P-Table, RPS File, and PF to produce an execution memory image.

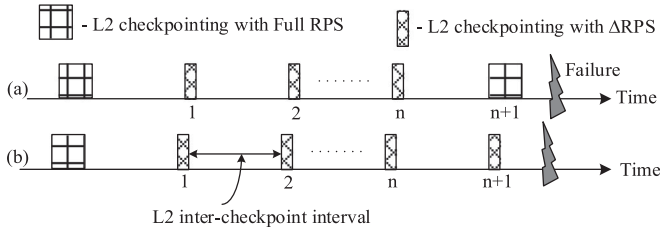

Fig. 7. Sequence of L2 checkpointing scenario.

distinct page number enters P-Table to establish a new entry, and the contents of the page are fetched for deduplication (as shown by the dashed arc in Fig. 5), one data pattern after another against those recorded in DeD-Table. A data pattern (say, x) is taken by Hash Engine (HE) to produce an index (say, h(x)) for accesses to DeD-Table, where h (x) is obtained by "hash(x) mod |DeD-Table|", with |DeD-Table| equal to the number of entries in DeD-Table and hash(x) being CRC-32. At the start of every coalesced batch, the contents of P-Table are inherent from the immediately previous batch, as P-Table records those dirty pages encountered so far, starting from the very first coalesced batch during job execution. In contrast, DeD-Table starts from scratch at the beginning of every deduplication batch.

For every matching page recorded in P-Table, DU first de-references α patterns pointers recorded in PF, starting at the location pointed by PF* of the same P-Table entry. Each pointer in PF references a data pattern in RPS File. The de-referenced α data patterns are the previous contents of the matched dirty page (in last coalescence batch of IC files), to be used for deduplicating current contents of the page at hand. For example, deduplicating Page 2 (i.e., the first entry of C-List) which also exists in P-Table, DU de-references α pointers from PF, starting at the index specified by PF* (i.e., 32), populates them in DeD-Table, as shown in Fig. 5b.

The duplicate data patterns are filtered out by DeD-Table. Each data pattern, say x, of a checkpoint file under deduplication, is probed against a candidate set in DeD-Table (indexed by h(x)) to see if it exists therein. If x does not exist, it is recorded in DeD-Table. In this case, the RPS* of the newly recorded entry denotes the position of x in Δ RPS File (say, k). If x exists, it is not to enter Δ RPS File, but the pointer of its matched entry is updated into Pointer File. For table utilization enhancement, DeD-Table follows a variant of Cuckoo hash [24] with two hash functions and 2-way set-associativity to manage its entries (not shown in Figure). After exhausting every pattern of all pages existing in C-List through the deduplication process, resulting PF (Pointer File), P-Table, and Δ RPS File denote a compressed version of the batch of IC files originally kept in L1 storage, and they replace those IC files therein. Additionally, the compressed version of the batch of IC files is sent to remote persistent storage for L2 checkpointing.

3.3 Execution State Restore

Execution state restore from failures takes two different paths, depending on the type of failures. For a permanent failure, all Δ RPS Files including the very first full RPS plus the most recent P-Table and PF are moved to the substitute core, where the execution state is reconstructed before execution resumes therein, as depicted in Fig. 6. Decompressing a page, which is recorded in a P-Table entry, is achieved by dereferencing α consecutive mapping pointers (from PF) that are located at the starting index specified by the PF* field of the recorded P-Table entry. This process repeats for every page existing in P-Table to complete execution state restore. On the other hand, a transient failure calls for restore locally, first by copying the page contents of all the pages in coalesced page list, which is obtained from L1 checkpoint files produced after the last L2 checkpoint and kept in local storage. In addition, those pages in the application memory footprint (recorded in P-Table) but are not present in the coalesced list, are reconstructed using the P-Table, PF, Δ RPS and a full RPS File as shown in Fig. 6. Without complex computation, REX has fast decompression, resulting in express restore.

3.4 Placement of Complete Coalescence

The remote checkpointing cost under incremental L2 checkpointing (i.e., *batched coalescence*, which works on a batch of ICs and generates Δ RPS File) is less than that of full L2 checkpointing (i.e., *complete coalescence*, which coalesces all ICs including the very first full checkpoint and deduplicates them to obtain a full RPS File, letting all prior checkpoint files discarded); however, the additional restore cost incurred by each Δ RPS File may negate all potential gains if the total number of Δ RPS Files exceeds a limit. Given that restoring to the latest execution state requires an early full RPS File plus all Δ RPS Files generated after the early file. Considering the checkpointing cost and the restart cost associated with each Δ RPS and full RPS File, REX keeps an L2 checkpoint in the form of full RPS (rather than Δ RPS), whenever the overall cost under Δ RPS is found to exceed that under full RPS.

Fig. 7 shows two different L2 checkpointing sequences (each of which involves $n+1$ intervals of L2 checkpointing): (a) very first full RPS followed by n Δ RPS and then a full RPS, and (b) very first full RPS followed by $(n+1)$ Δ RPS. Let the mean checkpointing cost and the restore cost associated with a Δ RPS File be C and δ, respectively, while those of full RPS are Ψ and γ. Let the probability of system failures within the next L2 checkpoint interval be denoted by p. Failures are assumed to occur randomly (following a Poisson distribution), with the mean time between failures (MTBF) of M, as commonly adopted earlier [4], [27], [28].

The following analysis aims to determine optimal selection of the coalescence type (i.e., batched or complete) for L2 checkpointing under REX. This selection process minimizes the overall checkpointing cost such that total execution time overhead is kept as low as possible. From Fig. 7a,

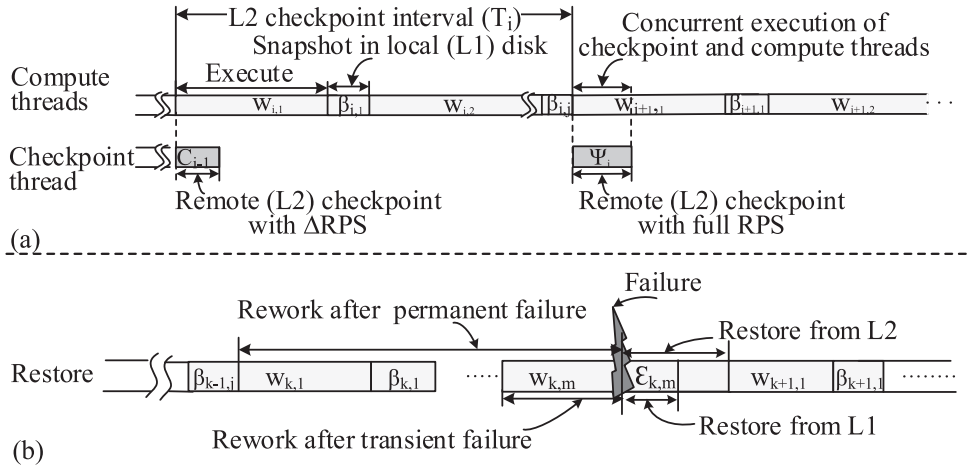

Fig. 8. Execution timing of MLC under REX, where (a) execution snapshot ("$\beta_{i,j}$") contributes to execution time penalty while remote checkpointing ("C_i or Ψ_i") are handled by a separate thread run concurrently during job execution ("$w_{i,j}$"), and (b) rework overhead and restore times after failures are shown.

if there is no failure before the next L2 checkpoin, the total checkpointing overhead cost (up to taking the next L2 checkpoint) equals the sum of all checkpointing costs (i.e., $\Psi + \mathrm{n} \cdot \mathrm{C} + \Psi$). To simplify our derivation, failures during L2 checkpointing are ignored, since they tend to affect proper selection of a coalescence type (i.e., batched or complete) negligibly. On the other hand, if a failure happens, besides all checkpointing costs, a restore cost (γ) is also incurred due to restoring from a full RPS file that was just taken (i.e., $\Psi + \mathrm{n} \cdot \mathrm{C} + \Psi + \gamma$). Thus, the expected overhead cost under a failure (with probability p) within the next L2 interval is given by

$$(1-p)\,(2 \cdot \Psi + \mathrm{n} \cdot \mathrm{C}) + p \cdot (2 \cdot \Psi + \mathrm{n} \cdot \mathrm{C} + \gamma). \tag{1}$$

Similarly, for Sequence (b) of Fig. 7, the expected overhead cost equals

$$(1-p)(\Psi + (\mathrm{n}+1)\mathrm{C}) + p \cdot (\Psi + (\mathrm{n}+1)\mathrm{C} + \gamma + (\mathrm{n}+1)\delta). \tag{2}$$

The invocation of a full RPS L2 checkpoint will minimize the overall cost, if Eq. (1) $\leq$ Eq. (2), namely,

$$\begin{aligned}&(1-p)(2 \cdot \Psi + \mathrm{n} \cdot \mathrm{C}) + p \cdot (2 \cdot \Psi + \mathrm{n} \cdot \mathrm{C} + \gamma) \\ &\leq (1-p)(\Psi + (\mathrm{n}+1)\mathrm{C}) + p(\Psi \\ &\quad + (\mathrm{n}+1)\mathrm{C} + \gamma + (\mathrm{n}+1)\delta).\end{aligned} \tag{3}$$

After algebraic manipulation, Eq. (3) gives rise to

$$\mathrm{n} \geq \frac{\Psi - \mathrm{C}}{p \cdot \delta} - 1.$$

With MTBF of M and the L2 inter-checkpoint interval of T, the value of p is given by 1- exp(-T/M) [28], [29]. Hence, by measuring Ψ, C, δ, and T for the system, REX optimally selects a coalescence type (i.e., batched or complete) for L2 checkpointing to minimize the overall execution time. The simulated results under different values of these parameters are presented in Sections 4 and 6.

3.5 REX Failure Model and Assumptions

Job execution with REX concurrent checkpointing involves a sequence of repeated events, as shown in Fig. 8a. The job is executed for $w_{i,j}$ seconds, followed by low-cost execution state checkpointing (i.e., taking an execution snapshot when job execution is halted and keeping the snapshot in local disk) for "$\beta_{i,j}$" seconds, during the i batch of events. This execution halt duration $\beta_{i,j}$ is typically small (i.e., $\beta_{i,j} \ll$ the inter-IC interval of "$w_{i,j}$"), as Page-cache hides the disk write latency [22] (see, Fig. 16) that makes snapshot taking fast (via memory copies). Job execution then resumes. REX takes IC several times locally (to produce a batch of IC files kept in L1 storage) before initiating a dedicated checkpointing thread to take one remote checkpointing to persistent L2 storage with the checkpointing cost of $C\,(\mathrm{or}\,\Psi)$ under batched (or complete) coaleesence, as depicted in Fig. 8a, where the L2 checkpoint interval (T_i) of the i batch of events equals $T_i = \Sigma j(w_{i,j} + \beta_{i,j})$.

Failures are assumed to occur randomly (following a Poisson distribution) with the MTBF of M. Two types of failures with known probabilities exist: transient and permanent failures. The former refers to ones that are not persistent and their involved compute cores may resume execution successfully after restore, whereas the latter refers to ones that render involved compute cores wholely unavailable. With 2-level MLC, execution can be restored from L1 checkpoints after transient failures. On the other hand, a permanent failure resorts to an L2 checkpoint and a substitute core for recovery.

3.5.1 Control of L2 Checkpointing

Recent articles [36], [37], [40] had pursued an L2 checkpointing analysis. However, REX cannot be analyzed in a similar way due to its four specific features, prompting the development of our REX analysis. First, L1 checkpointing under REX is adaptive (based on AIC), instead of being periodic with a fixed interval assumed in all earlier analyses. Secondly, the checkpointing cost (reflected by the checkpoint data volume size) is not a constant (see Fig. 3). Thirdly, REX employs a separate (dedicated) thread for handling L2 checkpointing (run concurrently with regular execution threads) to lower the job execution time penalty caused by checkpointing,

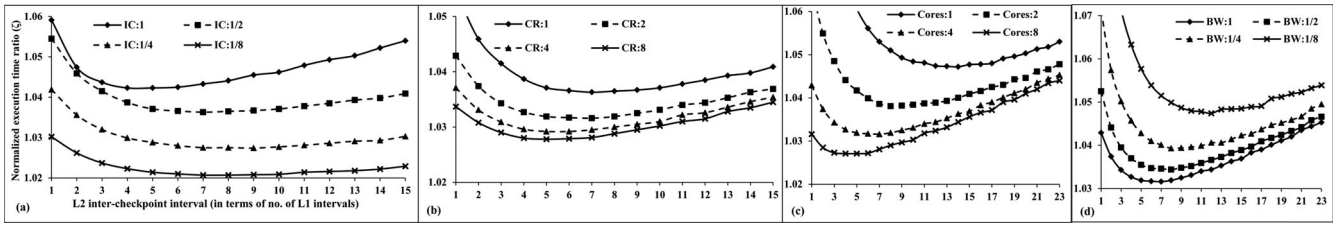

Fig. 9. Normalized execution time ratio (ζ) versus L2 inter-checkpoint intervals (in terms of the number of mean L1 checkpoint interval) of a long running job, where (a) denotes ζ for four IC amounts, with the other 3 parameters (CR, core count, and BW) held at fixed, default values. Similarly, (b) denotes ζ for four CR amounts, with the other 3 parameters held at fixed values, whereas (c) and (d) denote ζ for four core count amounts and four BW amounts, respectively.

instead of pausing execution threads for L2 checkpointing. Finally, our REX operates under incremental checkpoints, while all previous studies consider only full checkpoints.

With REX, the overall execution time of an application is determined by three factors (1) $\beta_{i,j}$ and $\mathrm{w}_{i,j}$ (2) $\varepsilon_{i,j}$ plus rework after a transient failure, and (3) C_i or Ψ_i and R_i plus rework after a permanent failure, where $\varepsilon_{i,j}$ and R_i refer respectively to the mean local restore time and the mean remote restore time, as illustrated in Figs. 8a and 8b. Mean rework after a transient (or permanent) failure is dictated by $\mathrm{w}_{i,j}$ (or T_i). Among the three time-determinant factors, only the last one is affected by L2 checkpointing control, because the sequence of L1 checkpoints is determined by AIC and $\varepsilon_{i,j}$ is related to $\beta_{i,j}$ (which is dictated by the number of dirty and new pages since the last L1 checkpoint). If L2 checkpointing time for Batch i of L1 checkpoints (T_i) is longer, mean rework after a permanent failure is larger, in favor of reducing T_i (with more frequent L2 checkpointing). On the other hand, the mean remote restore time R_i depends on the aggregate size of all L2 checkpointing files prior to the latest complete coalesce (i.e., full RPS), with the size dictated by the volumes of L1 checkpointing files involved and their compression ratios (CRs, whose details are given in Section 6.4.2) in those prior L2 checkpoints, as shown in Fig. 8b. This is because restore under incremental checkpointing involves k L2 checkpoint, for all $k < i$. The most desirable time point for taking L2 checkpointing exists, and it can be approximately estimated in runtime.

4 L2 Checkpointing Characteristics

For express restore while containing the overall execution time, we have conducted extensive event-driven simulation to gather the mean execution time of a long-execution job. The system under our simulation follows AIC [12] to produce L1 checkpoints, whose inter-checkpointing intervals are not fixed and observed to exhibit the coefficient of variance (defined as standard deviation divided by $|\mathrm{mean}|$) equal to 0.12. For example, with mean equals 120, L1 inter-checkpointing intervals (for one standard deviation) range from $105.6\,(=120\times0.88)$ to $134.4\,(=120\times1.12)$. Additionally, the simulated system has event parameters (see Fig. 8) as follows: the latency for taking a very first snapshot (L1 checkpoint time) of $\beta_{0,0}=2.0$, the mean L1 restore cost (from a full checkpoint) of $\varepsilon_{0,0}=20$, and the mean L2 checkpoint time and restore time (with full RPS File) of $\Psi_i=60$ and $\gamma=30$, respectively; while under Δ RPS File, L2 checkpoint time and restore time are dictated by the IC file sizes and CRs (compression ratios). Additionally, the system has a mean time between failure of $\mathrm{M}=$ 10,000 sec., with permanent failures accounting for 16 percent of all failures (as stated in an earlier study [3]). Recall that a permanent failure is recovered from the most recent L2 checkpoint, plus every prior L2 checkpoints (if any), since restore under any incremental checkpointing involves all prior checkpoints.

As depicted in Fig. 9, the normalized execution time ratio (ζ) versus the L2 inter-checkpoint interval (in terms of the number of the mean L1 checkpoint interval) is governed by a convex curve, where ζ refers to the execution time normalized with respect to that without checkpointing and without failures. Every ζ value shown in the figure is averaged over 10,000 event-driven simulation runs. The system is evaluated under four parameters (i.e., IC size versus full checkpoint size, CR, the core count per node involved in execution, and network bandwidth available for a node), which vary one at a time, with other three parameters held at default values. The default values of IC size versus full checkpoint size, CR, core count, and BW are 1/2, 2, 4, and 1, respectively. A large checkpoint volume refers to any application whose IC file sizes are comparable to its full memory footprint, with limited file compression (i.e., to exhibit low CRs). In contrast, a small checkpoint volume may result from applications with either relatively small IC files or high compression. It is found from Fig. 9 that ζ is higher for a larger checkpoint volume, since checkpointing (or restore) time overhead is dictated by the volume involved in checkpointing file writes to storage (or the file reads from storage plus data transfer).

As shown in Fig. 9a, ζ is minimum for the L2 inter-checkpoint interval near 4 (or 8) under the large IC size (or the small IC size). As shown in Fig. 9a. Having a smaller IC size and thus, a reduced L1 inter-checkpoint interval, REX waits for more L1 checkpoints before placing an expensive L2 checkpoint to lower the overall execution time. Similarly, Fig. 9b depicts that the desirable L2 inter-checkpoint interval reduces from 7 to 4 when CR rises from 1 to 8. Shrinking the checkpoint data volume, a higher CR lowers the L2 checkpointing cost, thus shortening the L2 interval to have a smaller rework time after a failure. Additionally, under the multicore system, REX employs a dedicated L2 checkpointing thread, which shares the available core(s) with regular execution thread(s), thus incurring a slight slowdown in job execution. The execution slowdown caused by L2 checkpointing is lightened as the number of cores grows. As a result, the preferred L2 inter-checkpointing interval drops from 13 (L1 checkpoints) to 5 when the number of cores grows from 1 to 8, as shown in Fig. 9c. Here, the total number of executing threads is assumed to equal the available cores. Furthermore, the impact of network bandwidth sharing between execution nodes and remote storage is

TABLE 1
Optimum L2 Inter-Checkpoint Interval (in Terms of No. of L1 Intervals) for a Various Combinations of IC and CR Values, Under $\mathrm{Cores} = 4$ and BW = 1

IC↓/CR→	1	2	4	8	12	16
1/16	9	7	7	7	6	5
1/12	8	6	6	5	5	5
1/8	8	7	6	5	5	5
1/4	8	7	6	5	5	5
1/2	7	7	6	5	5	5
1	5	4	4	4	4	4

shown in Fig. 9d. It reveals that the desirable L2 inter-checkpoint interval jumps from 7 (L1 checkpoints) to 12 to yield significant execution time overhead, if bandwidth decreases from 1 to 1/8 because of longer data transfer times.

In summary, the IC size is the only parameter that controls the L1 inter-checkpoint interval, while the L2 interval depends on all four parameters (i.e., IC size, CR, core count, and BW). Fig. 9 indicates that an increase in CR, core count, or BW shortens the preferred L2 interval. Under a smaller IC size, REX takes more frequent L1 checkpoints and gathers a larger batch of L1 checkpoints before placing an L2 checkpoint to reduce the overall execution time, mainly due to lowering the remote checkpointing cost.

4.1 Runtime L2 Checkpointing Control and Discussion

Our simulation evaluation of the execution time characteristics under MLC facilitates the development of runtime L2 checkpoint control. To identify the desirable timepoint for L2 checkpointing, it requires IC and CR values upon each L1 checkpoint to determine if an L2 checkpoint should be taken, for a given full memory footprint (of an application). The IC value is quickly determined by comparing the L1 checkpoint with the very first full checkpoint size, while the compression ratio (CR) obtained in the immediate prior L2 checkpoint is used. The approximate interval (χ) is then determined from the multivariate interpolation [23] using the data set presented in Table 1. The dataset represents the L2 inter-checkpoint interval (in terms of the number of L1 intervals) to yield the shortest overall execution time. The interval $[\chi] = f(IC, CR)$ is computed after each L1 checkpoint. If there have been at least $[\chi]$ L1 checkpoints taken so far since the last L2 checkpoint, an L2 checkpoint is then invoked, where $[\chi]$ is the nearest integer mapping function which maps a number to the range [4], [9]. This runtime control is adopted to produce checkpoint files for evaluating our REX performance, as detailed in Section 5.

5 Evaluation Methodology

Experimental evaluation has been performed on our testbed using various benchmarks. Benchmarks for evaluation are of three categories: single-threaded, multi-threaded, and multi-node runs, under the following hardware and system software settings.

5.1 Hardware and System Software

Our testbed consists of four Dell PowerEdge R610 servers, each with two quadcore Xeon E5530 processors that operate at 2.4 GHz and have 8-MB shared cache each. Every server runs 64-bit CentOS 5.5 with kernel version 2.6.18 and contains 32 GB of physical memory (with the page size of 4096 bytes), one 7200-RPM SATA disk with 512 GB, one 5400 RPM SATA disk with 2 TB, and one 512-GB SATA SSD. Additionally, the servers are connected through Fast Ethernet, Gigabit Ethernet, and 10-Gigabit Ethernet links for measuring data transfer rates as a function of file sizes over those links.

Adaptive incremental checkpointing (AIC) software is installed for the experiment to gather a full checkpoint file first, followed by a sequence of incremental checkpoint files. Meanwhile, for the comparison purpose, Xdelta3 [14] and the popular compression library of bzip2 [21] are installed, so is the page-aligned delta compressor (called Xdelta3-PA adopted in AIC). In addition, libhashchkpt [33] is implemented to process the incremental checkpointed files for comparison.

5.2 Applications and Evaluation Setup

Our evaluation targets single-threaded, multi-threaded, and multi-node applications, as listed in Table 1. The terms of applications and benchmarks are used interchangeably in the subsequent description. Single threaded applications include five benchmarks from the SPEC CPU2006 suite [19]. Each of them is processor-memory intensive and fits in 1-GB memory. Those benchmarks are chosen as representative applications of varying fields. SPEC provides a framework to run its benchmarks and to measure the results. Multi-threaded applications include five benchmarks from the PARSEC 3.0 suite, which represents workloads for programs with various emerging applications [20]. Meanwhile, to test the impact of large memory footprints of multi-node applications, six applications are chosen from NAS Parallel Benchmarks (NPB 3.3.1) [18]. All benchmarks selected from SPEC, PARSEC, and NPB 3.3.1 suites are compiled with the AIC checkpointing library [12].

For the evaluation purpose, a system with MTBF of $\mathrm{M} = 10{,}000$ seconds is assumed, where permanent failures account for 16 percent (as stated in an earlier study [3]). Under Linux, while the "kill -9" command gives a simple way to emulate a failure model, it is very tedious and time-consuming for extensive evaluation. To conduct our extensive evaluation, we instead measured the testbed parameters (i.e., disk read/write speed, network throughput) and compressor parameters (of compression/decompression rates, compression ratio) under REX and its counterparts. Note that REX counterparts are all incremental checkpoint (IC)-based, hence following AIC [12] (which determines the most suitable timepoints for L1 checkpoints) to produce a sequence of ICs. For comparing REX with its counterparts, we gathered a set of $\chi + 1$ checkpoint files (i.e., the first FC file followed by χ IC files) for each benchmark, where χ is the L2 inter-checkpoint interval in terms of the number of L1 intervals (as detailed in Section 4). The remaining course of benchmark execution is assumed to exhibit repeated sets of χ checkpoint files.

5.2.1 Summary of Evaluation Findings

A summary of our evaluation findings is provided below, with their details and discussion given in Section 6.

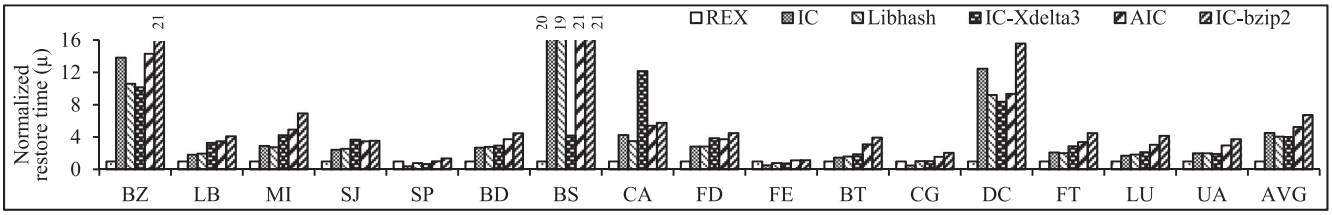

Fig. 10. Normalized restore time (averaged over 10,000 simulation runs, denoted by μ), taking the restore time under REX as the base, for various benchmarks.

- REX shortens the restore latency by a factor of 4.5×, 4.1×, and 5.3× than incremental checkpointing (IC), Libhash [33], and AIC [12] counterparts, respectively, and enjoys slightly lower execution time overhead. Specifically, the overall normalized execution time ratio is smaller under REX (of 1.029) than under IC (of 1.032), under Libhash (of 1.032), and under AIC (of 1.067).
- REX exhibits better scalability than all its IC-based counterparts.
- REX's de-referencing approach yields decompression rate of 400 MB/s, being multiple times higher than disk I/O and network throughputs measured on our testbed. The rate is 21×, 10×, and 4× faster than those of bzip2, AIC, and Xdelta3, respectively, promising express execution state restores from failures.
- REX's page-aware compression module gives rise to a comparable compression ratio of 2.4× (versus 1.4× by libhash, 1.8× by AIC, 2.6× by bzip2, and 3.0× by Xdelta3).

6 Results and Discussion

Evaluation results on execution time and restore latency are presented and discussed first, followed by REX scalability, with testbed measurements and REX deduplication presented at the end.

6.1 Execution Time and Restore Latency

In this section, our REX is compared against previously implemented IC-based methods (i.e., IC [38], AIC [12], and Libhash [33]) and two IC file compressors (i.e., IC-Xdelda3 and IC-bzip2), in terms of the normalized execution time ratio **(ζ)** (defined as the ratio of the overall execution time to the execution time without checkpointing and without failures). Eexecution restore under REX and its IC-based checkpointing counterparts typically involve all IC files plus the first full checkpoint file. As discussed in Section 3.1, REX keeps the coalesced set of deduplicated IC files in L1 and in L2 storage, while its counterparts keep IC files only in L1 storage and every FC file in both L1 and L2 storage. For extensive evaluation, we collected the set of $\chi + 1$ checkpoint files (involving the very first FC file plus subsequent χ IC files, taken under the AIC decision [12]) for each benchmark, followed by repeated sets of χ IC files until the end of its execution.

REX and its counterparts all deal with identical sets of checkpoint files for each benchmark and follow our runtime L2 checkpointing control (as presented in Section 4) to invoke remote checkpointing. At each L2 checkpoint timepoint, REX coalesces the just collected batch of χ ICs, deduplicates them, and stores the resulting files in both local and remote storages. While for every L2 checkpoint, On the other hand, each REX counterpart at every L2 checkpoint timepoint compresses (if required) FC before saving the results in local and remote storages.

The compressor of AIC involves ***both*** the very first full checkpoint file and the current full checkpoint file for improved compression, so do the compressors of Libhash and IC-Xdelda3. In contrast, IC-bzip2 compresses the current full checkpoint file itself using bzip2. Unlike REX, however, they do not conduct file coalescence. After compressing, every such checkpointing method transfers the compressed file to L2 storage remotely via a Gigabit Ethernet link in the secure mode (i.e., secure copy (***"scp"***)). For extensive evaluation, the testbed parameters (i.e., disk read/write speeds, network throughputs) and compressor features (i.e., compression ratios, compression and decompression speeds) were measured and presented in Figs. 15, 16, 17, 18, 19 in Section 6.4.

6.1.1 Restore Latency

Given that the time for failure restore is on the execution critical path, REX most concerns the restore latency upon failures. The restore latency depends on failure types. A transient (or permanent) failure incurs the total time spent to (1) read the checkpoint file(s) from L1 storage (or from connected L2 storage via the network link), (2) decompress the file(s) (if requaired), and (3) reconstruct the full system execution state out of the file(s) in a method-specific way, as follows. Under IC, AIC, Libhash, IC-Xdelta3, and IC-bzip2, the full system state is derived from ***all*** checkpoint files. On the other hand, REX conducts page-aware deduplication according to the coalesced list (if any), to involve *only the latest version of every page* with its prior version(s) all skipped for considerably shortened reconstruction times.

Normalized restore times (each averaged over 10,000 event-driven simulation runs, denoted by μ) for various benchmarks are illustrated in Fig. 10, where μ is defined as the ratio of the average restoration time under a checkpointing method (i.e., REX, IC, Libhash, AIC, IC-Xdelta3, or IC-bzip2) to that under REX for a given benchmark. For our evaluation, a multi-core node (with four cores shared by benchmark execution threads and the L2 checkpointing thread) is equipped with L1 storage and is connected through a dedicated Gigabit Ethernet link to remote L2 storage in support of MLC.

Normalized restore times are lower for 13 (out of 16) benchmarks under REX than under its counterparts, as shown in Fig. 10. Specifically, Benchmark BS under REX (see Figs. 17 and 18) is seen to involve small IC files (see Fig. 3) and to enjoy high compression and quick

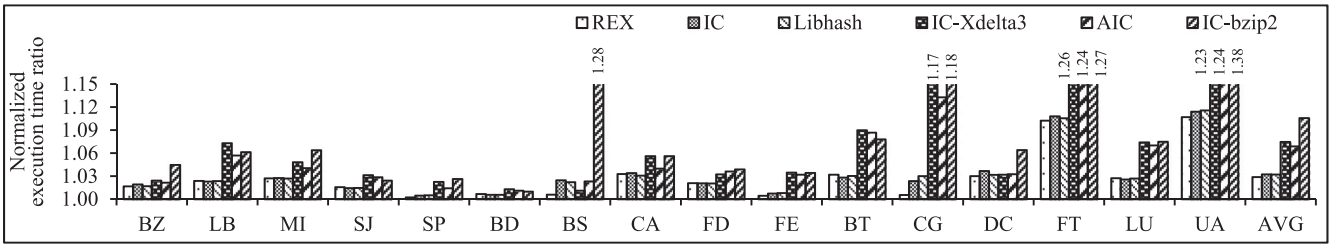

Fig. 11. Normalized execution time ratio (ζ) (the ratio of the overall execution time to the execution time without checkpointing and without failures).

decompression, exhibiting the largest restoration time gaps from those under its counterparts. REX restores from failures 19+ times quicker than IC, Libhash, AIC, and IC-bzip2; it is 4.2× faster than IC-Xdelta3. Compressed file sizes and decompression rates are main determinants for its express restore from L2 storage. For L1 restore (from transient failures), the total number of IC files, besides the compressed size and the decompression rate, plays a vital role on the total restore time. This is because upon arrival of every IC file, REX updates the coalesce list (C-List), which tracks the most recent version of pages resided in the IC file (see Fig. 5). By leveraging page-aware deduplication and C-List, REX fetches only the required data blocks to shorten the restore time after a failure. On the other hand, none of REX counterparts neither maintains C-List nor adopts page-aware decompression, thereby calling for all files to be read out fully and decompressed (if any) before system state reconstruction is undertaken. As Libhash, AIC, and IC all exhibit limited compression under BS, they require longer restore times than REX. On the other hand, IC-bzip2 lags behind because of its lower decompression rate. While IC-Xdelta3 has comparable CR and a similar decompression rate as those of REX, it lags behind REX in restoration due to its absence of C-List and page-aware decompression.

Meanwhile, REX achieves high CRs for BZ and DC to yield restore time reduction by 8× to 21× (from Fig. 10) when compared with other methods. Although three benchmarks (SP, FE, and CG) have small IC file sizes, μ under REX is higher than that under IC checkpointing because, for a smaller file size, REX waits for a longer time to invoke complete coalescence, which discards all prior checkpoint files resided in L1 and L2. Note that REX has slightly larger μ values than IC (or Libhash and IC-Xdelta3) checkpointing for the benchmarks of SP, FE, and CG (or SP and FE), due mainly to its limited compression gains under those benchmarks. Nonetheless, REX always enjoys the shortest overall execution times (which include the restore times after failurs) among all benchmarks for every benchmark examined (as will be seen in Fig. 11).

While LB has large IC file sizes (according to Fig. 3) and is subject to limited compression (see Fig. 17), REX still achieve the smallest μ among all checkpointing mechanisms. For such a benchmark, REX invokes file coalescence on every L2 checkpoint (explained in Section 3.4) to reduce the L2 restoration latency, while taking advantage of C-List and page-aware decompression to reconstruct the system states via local checkpointing files for express L1 restoration. The average μ values of all benchmarks, as shown in the rightmost group in Fig. 10, signify that REX is 4.5×, 4.1×, and 4.0× speedier in restore than its IC, Libhash, and IC-Xdelta3 counterparts, respectively. In addition, REX restores from failures 5.3× (6.7×) faster than AIC (or IC-bzip2).

6.1.2 Execution Time

We measured normalized execution times (averaged over 10,000 event-driven simulation runs) for various benchmarks using same sets of checkpoint files, system parameters, and checkpoint method-dependent parameters employed to gauge the restore latency.

The normalized execution time ratio **(ζ)** for various benchmarks under different checkpointing methods is depicted in Fig. 11. The figure shows that SP has the smallest ζ (of 1.002) under REX, since IC, Libhash, IC-Xdelta3, AIC, and IC-bzip2 lead to their ζ values of 1.005, 1.005, 1.023, 1.014, and 1.026, respectively. Having relatively small full checkpoint (FC) size of 41.5 MB and tiny IC file size of 1.25 MB (see Fig. 3 and Table 2), SP invokes a small checkpointing cost per checkpoint under REX, to favor a short checkpoint interval for lowered rework costs after failures, achieving the lowest overhead of 0.2 percent. On the other hand, REX's counterparts (i.e., AIC, IC-Xdelta3, IC-bzip2) spend significant amounts of times to compress (see Fig. 19) FC on every L2 checkpoint, resulting in higher overall execution times.

Similarly, REX exhibits smallest ζ values for those benchmarks with tiny IC file sizes (i.e., BS, FE, and CG), as can be found in Fig. 11. The execution times of those benchmarks are almost identical under IC as under Libhash, because compressing the involved IC files takes small times under Libhash. In general, Libhash and IC exhibit almost identical

TABLE 2
Target Benchmarks and Their Memory Footprints (MFs) in Pages of 4 KB Each

Benchmarks		MF (in pages)
SPEC CPU2006	bzip2 (BZ)-Compression & decompression	91791
	lbm (LB) - Fluid Dynamics	107213
	milc (MI) - Quantum Chromodynamics	173646
	sjeng (SJ) - Artificial Intelligence	46458
	sphinx3 (SP) - Speech recognition	10236
PARSEC	blacksholes (BS) - Portfolio	158699
	bodytrack (BD) - Computer Vision	9422
	cannel (CA) - Chip design	252142
	fluidanimater (FD) - Hydrodynamics	78895
	ferret (FE) -Content similarity search	26412
NPB 3.3.1	BT - Block Tri-diagonal solver	176962
	CG - Conjugate Gradient	123220
	DC - Data Cube	254555
	FT - Discrete 3D FFT	1320583
	LU - Lower-Upper Gauss-Seidel solver	149567
	UA - Unstructured Adaptive mesh	1791114

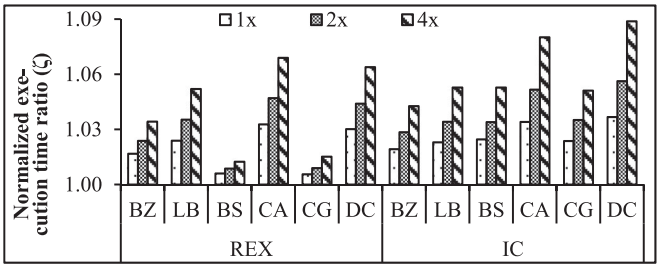

Fig. 12. Normalized execution time ratio (ζ) under various failure rates.

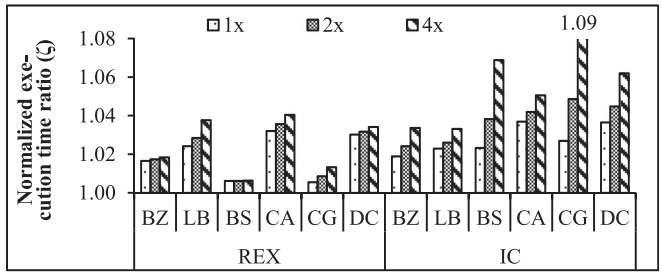

Fig. 13. Normalized execution time ratio (ζ) under various scale-sized I/O and network bandwidth.

overall execution times for any benchmark which has low compression under Libhash. Although BS enjoys high compression under IC-bzip2, the overall execution time is extremely high due to excessive compression time overhead (i.e., at the slow rate of ~2 MB/s). Similarly, with small IC file sizes but relatively large FC sizes and limited compression under all other compressors, Benchmark CG incurs very high execution time overhead of 17 percent for IC-Xdelta3, of 13 percent for IC-bzip2, and of 18 percent for AIC, in sharp contrast to 0.6 percent under REX. Benchmark FT (or UA) involves larger checkpoint files sizes (i.e., at least 3.5 GB each) to yield ζ of about 10 percent (or 11 percent) under REX, in comparison to 11 percent (or 12 percent) under both IC and Libhash, as depicted in Fig. 11. Benchmark UA has large checkpoint sizes, but their CR values are small and their decompression rates are low under IC-Xdelta3, AIC, and IC-bzip2, making their corresponding ζ values to stand respectively at 1.23, 1.24, and 1.38, far larger than 1.11 under REX.

Note that if the involved core count of a node increases from 4 to 8, the normalized execution time ratios of Benchmarks FT and UA reduce from $\sim$ 1.11 down to $\sim$ 1.09 under REX (not shown in Fig. 11). This reduction in ζ is mainly due to then hiding a larger portion of checkpoint time overhead (resulting from more cores) from the execution critical path.

With larger IC file sizes and small to no compression, Benchmarks LB, SJ and BT experience slightly longer overall execution times under REX than under IC. On an average, ζ under REX is 1.029, in contrast to 1.032, 1.032, 1.075, 1.069, and 1.106, respectively, under IC, Libhash, IC-Xdelta3, AIC, and IC-bzip2.

6.2 REX Scalability Evaluation

This section presents our evaluation results of REX scalability (in terms of higher failure rates and reduced network bandwidth). Six representative benchmarks are chosen for evaluation, covering various IC file sizes (as compared to full checkpoint size) and CR values. Specifically, chosen Benchmarks LB and DC belong to the large IC file size group, while BS and CG (or BZ and CA) are in the smaller (or moderate) IC size group. On the other hand, Benchmarks BZ, BS, and DC have high CR values, while CA (or LB) exhibits a moderate (or negligible) compression gain.

6.2.1 Impacts of Failure Rates

The overall execution time always extends as the failure rate increases. A higher failure rate desires more frequent checkpointing to curtail execution rework after failures, but more frequent checkpoints come naturally with higher checkpoint overhead, thereby longer overall execution times.

We measured the normalized execution time (averaged over 10,000 event-driven simulation runs) ratios for six representative benchmarks with various failure rates under REX and IC, as illustrated in Fig. 12. All simulation parameters, except MTBF, employed in this subsection are identical to those used in Section 6.1. Each group of bars under a benchmark in the figure represents ζ values for three system failure rates (i.e., MTBF of 10,000 seconds (as 1x), 5,000 seconds (as 2x), and 2,500 seconds (as 4x)).

As can be seen in Fig. 12, ζ surges as the failure rate hikes for every benchmark under REX and IC. For example, ζ for BZ under REX with the failure rate of 1x (or 2x and 4x) is 1.7 percent (or 2.4 and 3.4 percent), while the value under IC equals 1.9 percent (or 2.9 and 4.3 percent). Similarly, REX is subject to slightly lower execution time extension when the failure rate is doubled (or quadrupled) than IC for LB, BS, and DC, leading respectively in ζ of 1.1 percent (or 2.8 percent), 0.3 percent (or 0.6 percent), and 1.4 percent (or 3.4 percent), as opposed to 1.1 percent (or 3.0 percent), 0.9 percent (or 2.8 percent), and 2.0 percent (or 5.2 percent), respectively. Hence, REX is less vulnerable to failure rate hikes than IC.

6.2.2 Impacts of Node Count

The number of nodes that share network bandwidth for L2 checkpointing to remote shared storage plays a vital role on available network bandwidth available per node and thus I/O throughput per node. As the node count rises, the I/O throughput per node decreases with an increasing overall execution time overhead. A checkpointer that achieves high overall size reduction (i.e., smaller IC file sizes and/or better CR values) is to take a short time for compression and to incur a small extra execution time when I/O or network bandwidth drops.

Given MTBF of 10,000 seconds and slashing network and I/O bandwidth down to $\frac{1}{2}$ and $\frac{1}{4}$ of its original level, we rerun the event-driven simulation 10,000 times to measure the normalized execution time ratio (ζ), as demonstrated in Fig. 13. The bars referred by 1x, 2x, and 4x in the figure represent the ζ values under a network link and remote storage I/O shared by one node, two nodes, and four nodes, respectively.

Fig. 13 reveals that REX incurs additional execution time overhead for BZ, BS, and DC by 0.1 percent (or 0.2 percent), 0.0 percent (or 0.0 percent), and 0.2 percent (or 0.4 percent), respectively, when scaling down I/O and link bandwidth down to 1/2 (or 1/4). With same I/O and link bandwidth drops, however, IC suffers from more pronounced extra time overhead, by 0.5 percent (or 1.5 percent), 1.5 percent (or 4.6 percent), and 0.8 percent (or 2.5 percent), respectively

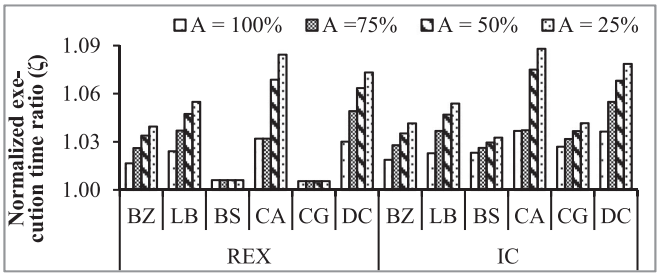

Fig. 14. Normalized execution time ratio (ζ) under various levels of memory available for deduplication.

for BZ, BS, and DC. The results confirm that compression could markedly lower the overall execution time. Meanwhile, REX is subject to slightly higher (1.2 percent) extra execution time overhead for LB under 4x than under 1x, due to limited compression then present, whereas IC experiences 1.0 percent extra time overhead. In addition, REX cuts down the overall execution time far more than IC for CG due to its small IC file sizes. In general, REX always outperforms IC, provided that file size reduction exists.

6.3 REX Overhead Assessment

Implemented on top of AIC [12], REX concurrent checkpointing involves overhead in the computing resource for deduplication and in memory (besides local and remote storage) for holding checkpoint-related data pages and deduplication files, as shown in Fig. 5. Given a job execution and L2 checkpointing (by a spawned thread) run concurrently, the deduplication process may interfere regular job execution more as the memory size available for deduplication is less, possibly resulting in an increase in the regular job execution time, when compared to that under an unlimited memory assumption.

To assess REX time overhead on regular job execution, we measured the normalized execution time (averaged over 10,000 event-driven simulation runs) ratios for six representative benchmarks with various memory sizes available for deduplication under REX and IC, as illustrated in Fig. 14. Except for available memory sizes, all simulation parameters adopted in this overhead assessment are identical to those used in Section 6.1. Each group of bars under a benchmark in the figure represents ζ values for four available memory sizes (i.e., $A = 100$ percent denoting memory available for deduplication equal to the memory footprint (MF) of an execution benchmark, $A = 75$ percent as memory available for deduplication equal to 75 percent MF, etc.).

With significantly smaller incremental checkpoint sizes under REX, Benchmarks BS and CG incur no additional execution time overhead even available memory is extremely scarce (of only 25 percent), as shown in Fig. 14. However, BS and CG under IC suffer from more execution time overhead (e.g., by 1.5 percent) as the available memory size drops (e.g., to $A = 25$ percent). From the figure, REX is seen to exhibit execution time overhead ranges from 3.2 percent (under $A = 100$ percent) to 8.5 percent (under $A = 25$ percent) for CA benchmark, and from 4.9 percent (under $A = 100$ percent) to 7.3 percent (under $A = 25$ percent) for DC benchmark. In contrast, IC yields execution time overhead ranges from 3.7 percent (under $A = 100$ percent) to 8.8 percent (under $A = 25$ percent) for CA benchmark, and from 5.5 percent (under $A = 100$ percent) to 7.9 percent (under $A = 25$ percent) for DC benchmark. Meanwhile, Benchmarks BZ and LB have their ζ values to stay nearly the same under REX as under IC over all available memory sizes examined, with a larger ζ for a small A.

Although IC does not involve any compression overhead, its ζ values hike for all benchmarks when the size of memory available for deduplication (which serves as temporary storage to keep an execution snapshot) shrinks, due to its increased L1 checckpointing time overhead, because a regular job execution then halts longer in order to finish writing its L1 checkpoint files (of a snapshot), resulting in higher execution time overhead.

6.4 Testbed and REX Deduplication

The objective of this section is to determine the testbed parameters (i.e., data read/write speed, network throughput) and compressors attributes employed by REX and its counterparts.

6.4.1 Testbed Measurement

We measured the maximum data transfer rates over Fast Ethernet, Gigabit Ethernet, and 10-Gigabit Ethernet links between a pair of connected nodes, by transferring various sized files, ranging from 2 MB to 4 GB, multiple times from one node to a connected node in secure and non-secure modes. Our secure mode used the "***scp***" (secure copy) command, while the non-secure mode employed the "***nc***" (Netcat) command. In addition, we measured the highest possible I/O throughputs of a fresh 2-TB HDD with 5400 RPM and a fresh 512-GB SSD (both having the ext3 file system) in contention-free environments under file sizes ranging from 2 MB to 4 GB. The disk write speed was measured after executing the *sync* command while Page-cache [22] was enabled, found to be always greater than 512 MB/s for both disks. In Linux, Page-cache hides the data write latency by utilizing the unused memory area as temporary storage (i.e., the disk buffer) and achieves a speed comparable to that of memory copies. However, for data reads, Page-cache does not reduce the time for the very first read of a file. The *sync* command forces all dirty data in memory to be written to the disk. The disk read speed under a file size was obtained by reading multiple same-sized files (first read), and the shortest read time of them was adopted to determine the maximum read throughput from the disk.

The measured throughput results of data transfer are depicted in Fig. 15, where the throughputs for both Gigabit and 10-Gigabit Ethernets under scp increase gradually as the file size rises from 2 MB to 128 MB before flattening out, to achieve the maximum rate of 46 MB/s. Similarly, nc exhibits a gradual (or marked) rise in the transfer rate up to 32 MB (or 128 MB) for Gigabit (or 10-Gigabit) Ethernet, with the peak rate of 112 MB/s (or 241 MB/s). Over Fast Ethernet, however, both scp and nc have an almost constant throughput (of 11 MB/s) for all file sizes.

Disk I/O speeds measured for a wide range of file sizes under HDD and SSD are demonstrated in Fig. 16. It is observed that the HDD (or SSD) read speed increases monotonically from the file size of 2 MB to 16 MB (or up to 64 MB) before saturating thereafter to around 47 MB/s (or 141 MB/

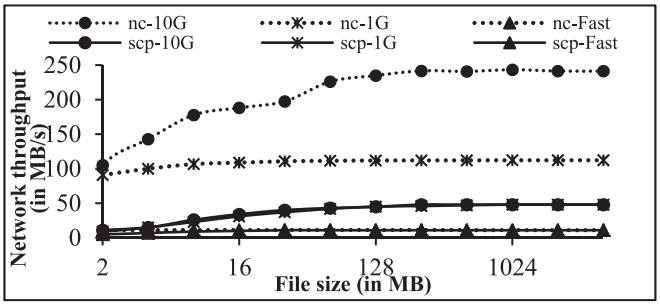

Fig. 15. Network throughput of different links measured in our testbed under various file sizes in both secure and non-secure modes of data transfer.

s). The write speeds (with Page-cache enabled) on both disks always exceed 512 MB/s for all tested file sizes.

6.4.2 REX Deduplication and Comparative Results

The REX's deduplication module is designed to satisfy the page level compression/decompression such that a page can be independently decompressed. As per our knowledge, there is no such compressor that works on a page boundary. Thus, for evaluation, we contrasted our REX deduplication with popular compression libraries (e.g., bzip2 [21] and Xdelta3 [14]) in terms of the compression ratio (CR), compression rate, and decompression rate. Additionally, page-aligned delta compressor (Xdelta3-PA) implemented using Xdelda3 library under AIC [12] as AIC and hash-based compressor used by libshashchkpt [33] as Libhash are also compared. The compression ratio denotes the ratio of the aggregate data volume of all original checkpoint files to that of all compressed checkpoint files for a given benchmark.

Deduplication Effectiveness. The effectiveness of deduplication (or compression) is reflected by its resulting CR (compression ratio). The CR results of different compressors for all benchmarks are depicted in Fig. 17, where the bars in the rightmost group indicate the GM values for the five compressors. From the figure, REX is seen to have its CR varying widely from 96 (for BS) to 1.05 (for BT). Benchmarks BT, LU display almost no compression gain for all compressors. Although bzip2 shrinks its search window down to as small as three bytes for compression [21], REX outperforms bzip2 in 5 out of 16 benchmarks (i.e., with higher CR values), signifying benefits due to global duplicate patterns (i.e., patterns shared across checkpoint files) under REX. Being a page-aligned delta compressor, AIC has to identify identical page numbers from the immediate prior checkpoint file (if any) for compression. Naturally, AIC cannot compress the very first checkpoint file, making its overall CR value smaller than that

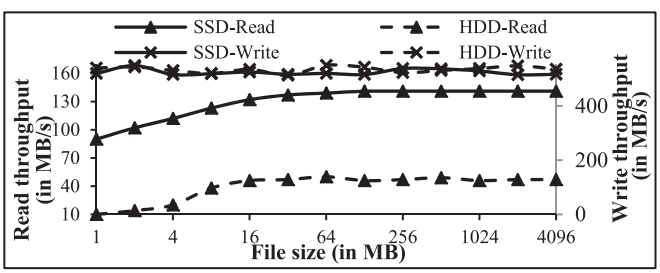

Fig. 16. I/O throughput of different storages measured in our testbed under various file sizes.

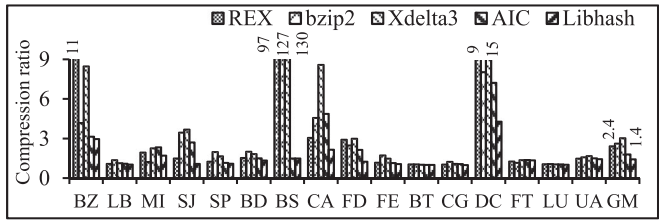

Fig. 17. Compression ratio (CR) results, normalized w.r.t. the original checkpoint data volume for various benchmarks.

of Xdelta3, even though it employs the Xdelta3 library. With a relatively larger block size (of 256 bytes), Libhash exhibits the worst compression ratio for all benchmarks, except MI and FT. Note that while a smaller data block size (e.g., 128, 64, or 32 bytes) may be adopted for Libhash, it results in higher overhead due to the fixed signature (of say, 16 bytes per data block for MD5 digest) maintained for each data block, rendering an inferior compression ratio.

REX employs fixed block size (of 32 bytes) and shares the duplicate patterns across the checkpointed files. This sharing advantage is revealed from the results of BZ, MI, FD, DC, FT, and LU, in which REX enjoys higher CR values than its bzip2 counterpart (which conducts local deduplication only, despite its use of overhead-heavy varying length data patterns). The mean CR of REX over all benchmarks is 2.4, in contrast to 2.6, 3.0, 1.8, and 1.4, respectively, of bzip2 Xdelta3, AIC, and Libhash.

Similarly, the weighted average decompression rates under various benchmarks for REX are illustrated in Fig. 18. From the figure, bzip2 is found to have the slowest decompression for all benchmarks except CA, for which Xdelta3 takes the longest time. Note that compression and decompression rate results under Libhash are almost constant (i.e., ~200 and 500 MB/s, respectively) in our testbed for all benchmarks, as demonstrated in the figure. Its compression rate mainly depends on the hash (say, MD5) rate for given block size (of 256 bytes), while decompression simply dereferences data patterns from the prefetched buffer.

As expected, REX has far speedier decompression than bzip2 for all benchmarks, mostly due to variable data pattern sizes under bzip2. The GM value of decompression rates of all benchmarks for REX is 401.3 MB/s, which is 20.9×, 10.1×, and 4.1× faster than those of bzip2, AIC, and Xdelta3, respectively.

Separately, the lowest decompression rate of REX (equal to 315.0 MB/s for FD) is many times faster than the maximum HDD (or SSD) read speed, which stands around 47 MB/s (or 141 MB/s) (see, Fig. 16), indicating that its data decompression time is well dwarfed by the disk read time during execution state restore.

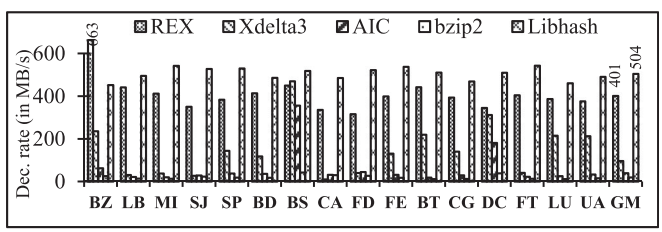

Fig. 18. Weighted average decompression rates for various benchmarks (with REX under the pattern length of 32 bytes).

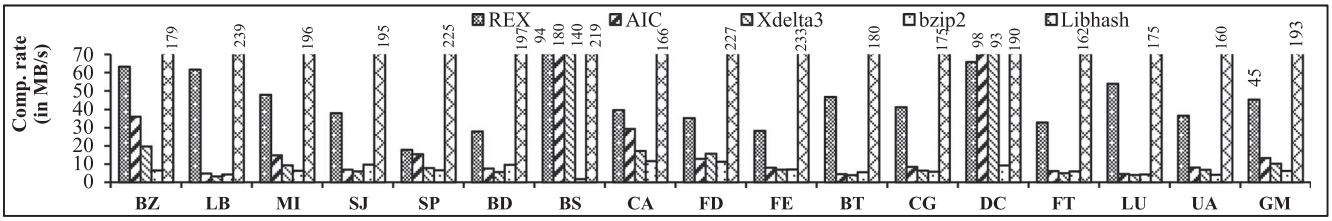

Fig. 19. Weighted average compression rate results for various benchmarks.

Additionally, Fig. 19 shows the data compression rates (in megabytes per seconds, MB/s) under various benchmarks. From the figure, the compression rate of REX for BZ equals 63.3 MB/s, whereas those for bzip2, Xdelta3, and AIC are 6.6, 19.7, and 36.1 MB/s, respectively, denoting that REX is 9.6×, 3.2× and 1.8× faster than bzip2, Xdelta3, and AIC, respectively. Having the largest compression rates for REX (of 94.0 MB/s), AIC (of 180.0 MB/s), and Xdelta3 (of 140.0 MB/s) but the smallest rate for bzip2 (of 1.8 MB/s), the benchmark BS reveals that the compression rate is algorithm-dependent.

The geometric mean (GM) of compression rates for all benchmarks under REX is 45.2 MB/s, representing 7.12×, 4.40×, and 3.37× faster than bzip2, Xdelta3, and AIC, respectively. Note that the measured compression and decompression rates of bzip2 using our testbed closely match those reported earlier [35].

7 Related Work

Various checkpointing mechanisms have been pursued, aiming originally at fault tolerance for long-running jobs [3], [11]. Their primary concern has been mostly on reducing storage time overheads associated with checkpointing [6], [7], [8], [10], [28], [39]. To this end, incremental checkpointing (IC) has been commonly adopted [7], [8], [11], [12], [30] for overhead reduction, by saving only modified and new (due to dynamic allocation) memory pages into the checkpoint files. Meanwhile, adaptive checkpointing for fault tolerance was considered earlier by either adjusting the checkpointing interval dynamically during task execution [9] or omitting checkpoints when the system was predicted to be safe [13]. Further, IC has been enhanced by either skipping checkpoints dynamically (based on the expected recovery time) [10] or computing the optimal block boundaries dynamically (rather than the fixed page boundaries) via a secure hash function, based on the history of changed blocks [11]. FENCE dynamically estimates system vulnerability in order to take appropriate actions [31]. Earlier hash-based IC compares the results of hashed data blocks with those of the immediate prior version, to decide if data blocks have been modified [11], [33], [34].

Recent adaptive IC (AIC) [12] predicts the most suitable points of time to do checkpoints, considering mean time before failure (MTBF) and smallest checkpoint latency possible, mainly focusing on the similarity degree of memory contents with respect to those of the prior checkpoint. It yields a checkpoint file which includes only those dirty and new pages.

MLC (multi-level checkpointing) can withstand a failure that renders one node inaccessible entirely (including its associated local storage) [3], [4], [6], [32], making it possible to greatly enhance job execution resilience. MLC does not perform every checkpointing to all storage levels because high-level storage (beyond the local, L1 level) is typically remote and shared. Instead, it takes more frequent checkpoints to less expensive L1 storage while checkpointing less frequently to more resilient high-level storage [3], [4]. Besides local storage overhead involved, MLC also consumes network bandwidth for data transfer to checkpoint files in remote nodes and/or shared storage. Recent articles analyzed the desirable L2 inter-checkpoint interval (as a function of L1 intervals) under a given failure rate and constant costs for checkpointing and restore [36], [37], [40].

8 Conclusion

Checkpointing takes the job execution states repeatedly and keeps them in storage as checkpoint files. It permits job execution recovery from failures using the prior checkpoint file (s). A restore-express (REX) strategy for MLC has been addressed to accelerate execution restore (after failures) drastically when compared with all earlier IC-based counterparts, while enjoying execution time overhead reduction slightly. With adaptive IC (introduced earlier [12]) to guide its first-level (L1) checkpointing, REX follows our developed runtime control for the second-level (L2) checkpointing at desirable time points, aiming to accelerate failure restore while keeping the overall execution time near the smallest possible. For effective L2 checkpointing (to remote, persistent storage), the batch of IC files produced since the last L2 checkpoint is coalesced before deduplication using data patterns existing not only within the coalesced L2 file but also across earlier L2 checkpoints. This is made possible by taking advantage of two unique insights for overhead reduction. Employing a dedicated L2 checkpointing thread to run concurrently with the execution threads in the multi-core system, REX limits the execution time overhead below 3.0 percent. According to the evaluation results obtained on our testbed under 16 benchmarks from SPEC, PARSEC, and NPB suites, REX is found to lower the mean restore latency by a factor of 4.5× and 4.0×, when compared with its baseline incremental checkpointing counterpart and Libhashchkpt. This proposed REX strategy applies to any system-level incremental checkpointing, be periodic or adaptive, for accelerating its execution restore after failures.

Acknowledgments

This work was supported in part by National Science Foundation under Award Number: CNS-1527051.

References

[1] G. Zheng, X. Ni, and L. V. Kalé, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *Proc. 2nd Workshop Fault-Tolerant HPC Extreme Scale*, Jun. 2012, pp. 1–6.

[2] D. Dirk Vogt, et al., "Lightweight memory checkpointing," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, Jun. 2015, pp. 474–484.
[3] A. Moody, G. Bronevetsky, et al., "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proc. IEEE/ACM Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Nov. 2010, pp. 1–11.
[4] N. H. Vaidya, "A case for two-level recovery schemes," *IEEE Trans. Comput.*, vol. 47, pp. 656–666, Jun. 1998.
[5] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, Oct. 1998.
[6] K. Sato, et al., "Design and modeling of a non-blocking checkpointing system," in *Proc. IEEE/ACM Conf. High Perform. Comput. Netw. Storage Anal.*, Nov. 2012, pp. 1–10.
[7] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *J. Phys.: Conf. Series*, vol. 46, pp. 494–499, 2006.
[8] M. Vasavada, et al., "Comparing different approaches for incremental checkpointing: the showdown," in *Proc. Linux Symp.*, pp. 69–80, 2011.
[9] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, Mar. 2003, pp. 918–923.
[10] S. Yi, J. Heo, Y. Cho, and J. Hong, "Adaptive page-level incremental checkpointing based on expected recovery time," in *Proc. ACM Symp. Appl. Comput.*, Apr. 2006, pp. 1472–1476.
[11] S. Agarwal, et al., "Adaptive incremental checkpointing for massively parallel systems," in *Proc. 18th Annu. Int. Conf. Supercomputing*, Jun./Jul. 2004, pp. 277–286.
[12] I. Jangjaimon and N.-F. Tzeng, "Adaptive incremental checkpointing via delta compression for networked multicore systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2013, pp. 7–18.
[13] A. J. Oliner, L. Rudolph, and R. K. Sahoo, "Cooperative checkpointing: A robust approach to large-scale systems reliability," in *Proc. 20th Int. Conf. Supercomputing*, Jun. 2006, pp. 14–23.
[14] J. MacDonald, "File system support for delta compression," *M.S. Thesis*, Univ. of California, Berkeley, May 2000.
[15] Y. Li and Z. Lan, "A fast restart mechanism for checkpoint/recovery protocols in networked environments," in *Proc. 38th Int. Conf. Depend. Syst. Netw.*, Jun. 2008, pp. 217–226.
[16] N. Bessho and T. Dohi, "Comparing checkpoint and rollback recovery schemes in a cluster system," in *Proc. 12th Int. Conf. Algorithms Archit. Parallel Process.*, Sept. 2012, pp. 531–545.
[17] A. Tridgell, "Efficient algorithms for sorting and synchronization," Ph.D. Dissertation, Australian National Univ., Canberra, 2000.
[18] NASA Advanced Supercomputing Division, "NAS parallel benchmarks," Mar. 2016. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html, last accessed July 2017.
[19] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sep. 2006.
[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Tech.*, Oct. 2008, pp. 72–81.
[21] J. Seward, "bzip2 and libbzip2, version 1.0.5: A program and library for data compression," 2007, [Online]. Available: http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.pdf, last accessed July 2017.
[22] R. V. Riel, "Page replacement in Linux 2.4 memory management", in *Proc. USENIX Annu. Tech. Conf.*, 2001, pp. 165–172.
[23] J. Meinguet, "Multivariate interpolation at arbitrary points made simple," *J. Appl. Math. Phys.*, vol. 30, no. 2, pp. 292–304, Mar. 1979.
[24] B. Fan, et al., "Cuckoo filter: Practically better than bloom," in *Proc. 10th ACM Int. Conf. Emerging Netw. Experiments Technol.*, Dec. 2014, pp. 75–88.
[25] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploring unconventional benefits from memory compression," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, Feb. 2014, pp. 638–649.
[26] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing content similarity to improve I/O performance," *ACM Trans. Storage*, vol. 6, no. 3, Sep. 2010, Art. no. 13.
[27] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *Proc. 44th IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, Jun. 2014, pp. 25–36.
[28] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Comput. Syst.*, vol. 22, pp. 303–312, 2006.
[29] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, pp. 530–531, 1974.
[30] R. Gioiosa, et al., "Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers," in *Proc. IEEE/ACM Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Nov. 2005, pp. 9–23.
[31] X.-H. Sun, Z. Lan, Y. Li, H. Jin, and Z. Zheng, "Towards a fault-aware computing environment," in *Proc. High Availability Perform. Workshop*, Mar. 2008.
[32] T. Z. Islam, et al., "mcrEngine: A scalable checkpointing system using data-aware aggregation and compression," in *Proc. Int. Conf. High-Perform. Comput. Netw. Storage Anal.*, Nov. 2012, pp. 1–11.
[33] K.B. Ferreira1, et al., "libhashckpt: Hash-based incremental checkpointing using GPU's," in *Proc. 18th Eur. MPI Users' Group Conf. Recent Adv.*, Sep. 2011, pp. 272–281.
[34] T. Knauth and C. Fetzer, "Vecycle: Recycling vm checkpoints for faster migrations," in *Proc. 16th Annu. Middleware Conf.*, Nov. 2015, pp. 210–221.
[35] Jarrod, "Linux RootUsers, Gzip vs Bzip2 vs XZ performance comparison's," Sep. 2015. [Online]. Available: https://www.rootusers.com/gzip-vs-bzip2- vs-xz-performance-comparison, last accessed July 2017.
[36] A. Benoit, A. Cavelan, Y. Robert, and H. Sun, "Optimal resilience patterns to cope with fail-stop and silent errors," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2016, pp. 202–211.
[37] A. Benoit, A. Cavelan, V. Le Fevre, Y. Robert, and H. Sun, "Towards optimal multi-level checkpointing," *IEEE Trans. Comput.*, vol. 66, no. 7, Jul. 2017, pp. 1212–1226.
[38] H. Li, L. Pang, and Z. Wang, "Two-level incremental checkpoint recovery scheme for reducing system total overheads," *PLoS One*, vol. 9, no. 8, Aug. 2014, Art. no. e104591.
[39] Z. Chen, J. Sun, and H, Chen, "Optimizing checkpoint restart with data deduplication," *Sci. Program.*, vol. 2016, Jun. 2016, Art. no. 10.
[40] S. Di, Y. Robert, F. Vivien, and F. Cappello, "Toward an optimal online checkpoint solution under a two-level hpc checkpoint model," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 244–259, Jan. 2017.

Purushottam Sigdel received the BE degree in electronics and communication engineering, in 2000, the MS degree in information and communication from Trubhuvan University, Kathmandu, Nepal, in 2006, and the MS degree in computer science from the University of Louisiana, Lafayette, in 2014. He is working toward the PhD degree from Center for Advanced Computer Studies, the University of Louisiana, Lafayette. From 2000 to 2012, he worked as an instructor in the department of electronics and computer engineering in Pulchowk campus, Lalitpur, Nepal. His research interests include data compression, distributed computing, storage systems, and systems architecture.

Nian-Feng Tzeng (M'86-SM'92-F'10) has been with Center for Advanced Computer Studies, School of Computing and Informatics, the University of Louisiana, Lafayette, since 1987. His current research interest is in the areas of high-performance computer systems, and parallel and distributed processing. He was on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems*, 1998-2001, and on the editorial board of the *IEEE Transactions on Computers*, 1994-1998. He was the chair of Technical Committee on Distributed Processing, the IEEE Computer Society, from 1999 till 2002. He is the recipient of the outstanding paper award of the 10th IEEE International Conference on Distributed Computing Systems, May 1990, and received the University Foundation distinguished professor Award in 1997.